JPlatform 10 Documentation

Gestion des alarmes

JPlatform utilise en interne des tâches planifiées pour déclencher des actions à une date donnée. Par exemple, lorsqu'un contenu reçoit une date de publication et passe dans l'état planifié, une alarme est enclenchée pour gérer son passage à l'état publié. Cette gestion d'alarme est également utilisée avec les Worklow Express mais aussi pour envoyer les notifications ou encore pour générer le rapport du journal des accès.

L'API de JPlatform vous permet de programmer vos propres tâches planifiées en utilisant le package com.jalios.jdring.

1. L'API JDring

JDring est un package qui gère des planifications d'alarmes d'une manière similaire aux commandes cron et at des environnements Unix. Les alarmes peuvent être ajoutées dynamiquement, dans n'importe quel ordre, et peuvent être répétitives ou non. JDring a été conçu pour gérer de grande quantité d'alarmes sans que les performances de JPlatform soient dégradées. JDring n'utilise qu'une seule thread qui s'endort jusqu'à la prochaine alarme (via la méthode wait(time)).

L'API de JDring comporte principalement 3 classes :

- AlarmEntry : cette classe contient les paramètres de l'alarme ;
- AlarmListener : cette interface doit être implementée par les objets qui doivent être notifiés lors du déclenchement d'une alarme ;
- TransactionalAlarmListener : cette classe abstraite doit être surchargée par les objets qui doivent être notifiés lors du déclenchement d'une alarme et qui effectuent un traitement en relation avec JcmsDB ;
- AlarmManager : cette classe gère les AlarmEntry. Elle permet d'ajouter et de retirer des alarmes.

Pour gérer une nouvelle alarme, il suffit de créer une `AlarmEntry` que l'on ajoute à un `AlarmManager` . L' `AlarmEntry` doit contenir un `AlarmListener` qui sera invoqué lorsque la date de l'alarme sera atteinte.

1.1. La classe AlarmEntry

La construction d'une `AlarmEntry` peut se faire via plusieurs constructeurs :

- **public AlarmEntry(Date date, AlarmListener listener)**
  Construit une alarme pour une date donnée.
- **public AlarmEntry(int delay, boolean isRepetitive, AlarmListener listener)**
  Construit une alarme (éventuellement répétitive) pour une durée donnée
- **public AlarmEntry(int minute, int hour,int dayOfMonth, int month, int dayOfWeek, int year, AlarmListener listener)**
  Construit une alarme pour une planification précise et éventuellement répétitive
- **public AlarmEntry(String schedule, AlarmListener listener)**
  Construit une alarme pour une planification exprimée selon le format (simplifié) de la commande cron :
  `[minute] [heure] [jour du mois] [mois] [jour de la semaine] [année]`
  Les astérisques (*) spécifient que l'alarme doit être déclenchée de façon répétitive sur cette période.
  Les jours de la semaine commencent par le dimanche et leurs valeurs vont de 1 à 7.
  Exemples :
  - `30 10 * * * *` : tous les jour à 10h30
  - `30 10 1 * * *` : tous les 1er du mois à 10h30
  - `30 10 * * 2 *` : tous les lundi à 10h30

1.2. L'interface AlarmListener

Un `AlarmListener` doit implémenter la méthode `handleAlarm()` qui est invoquée lorsque l'alarme se déclenche. Elle reçoit en paramètre l' `AlarmEntry` correspondant.

Exemple :

```
public class MyScheduledTask implements AlarmListener {
  public void handleAlarm(AlarmEntry entry) {
    System.out.println("Wake up !");
  }
}
```

1.3. La classe TransactionalAlarmListener

Lorsque le traitement effectué au déclenchement de l'alarme nécessite des interactions avec la base de données JcmsDB, il doit être encapsulé dans une transaction. Cela concerne les traitements effectuant des recherches ou des écritures dans JcmsDB mais aussi les écritures dans JStore qui sont bien souvent accompagnées de requêtes à JcmsDB.

La classe abstraite `TransactionalAlarmListener` offre cette encapsulation. Il suffit simplement de dériver de cette classe et d'implémenter la méthode `handleTransactionalAlarm()` .

Exemple :

```
public class MyDBScheduledTask extends TransactionalAlarmListener {
  public void handleTransactionalAlarm(AlarmEntry entry) {
    System.out.println(channel.getDataCount(ArchivedPublication.class) + " archive(s) stored i
  }
}
```

1.4. La classe AlarmManager

L'enregistrement des alarmes se fait au près d'un `AlarmManager` via la méthode `addAlarm()` . Pour obtenir un `AlarmManager` , il est recommandé de passer par la méthode `channel.getCommonAlarmManager()` . Cette méthode gère la majorité des `AlarmManager` utilisés dans JPlatform. Elle garantit en particulier l'arrêt correct de ces `AlarmManager` (et de leur thread) lorsque le site redémarre.

Il est aussi possible d'utiliser son propre `AlarmManager` . L'utilisation d'un `AlarmManager` dédié permet de faire des opérations supplémentaires, comme par exemple la suppression de toutes les alarmes (ce qu'il ne faut surtout pas faire sur le gestionnaire commun). En contre partie, une nouvelle thread est créée. Pour obtenir un `AlarmManager` , il faut utiliser la méthode `channel.getAlarmManager(String key)` . L'attribut key représente le nom de l' `AlarmManager` .

2. Intégration dans JPlatform

2.1. Intégration programmatique

L'ajout d'alarmes dans un module se fait en utilisant un ChannelListener qui permet d'effectuer des actions au démarrage de JPlatform. Pour être pris en charge par le gestionnaire de module, le `ChannelListener` doit aussi implémenter l'interface PluginComponent.

Exemple de code :

```
public class MyChannelListener extends ChannelListener implements PluginComponent {

  public boolean init(Plugin plugin) {
    return true;
  }
  public void initAfterStoreLoad() {
    try {
      String schedule = "30 10 * * 2 * "; // Every Monday at 10h30
      AlarmListener alarmListener = new MyAlarmListener();
      AlarmEntry alarmEntry = new AlarmEntry(schedule, alarmListener);
      AlarmManager alarmMgr = Channel.getChannel().getCommonAlarmManager();
      alarmMgr.addAlarm(alarmEntry);
    }
    catch(com.jalios.jdring.PastDateException ex) {
      ex.printStackTrace();
    }
  }
  public void handleFinalize() {  }
  public void initBeforeStoreLoad() {  }
}
```

La déclaration du `ChannelListener` dans le fichier `plugin.xml` se fait par la balise `<channellistener>`.

```
<plugincomponents>
  <channellistener class="com.package.MyChannelListener" />
</plugincomponents>
```

2.2. Intégration déclarative

Il est possible de déclarer un alarmListener directement dans le fichier `plugin.xml`.

```
<plugincomponents>
    <alarmlistener class="com.package.MyAlarmListener" schedule="30 14 * * * *" manager="MyAla
</plugincomponents>
```

Alternativement, on peut préciser une fréquence (en minutes) au lieu d'utiliser une syntaxe à la cron :

```
<plugincomponents>
    <alarmlistener class="com.package.MyAlarmListener" frequency="30" manager="MyAlarmManager"
</plugincomponents>
```

Si le manager n'est pas précisé, alors un `AlarmManager` commun à tous les modules est utilisé.

3. Un exemple complet

Cet exemple met en place un suivi régulier de la consommation mémoire et le déclenchement d'alertes lorsque la quantité de mémoire devient critique.

Une première alarme, `MemoryInfo`, est programmée pour envoyer par mail l'état de la mémoire tous les jours à 9h30 et 14h30.

Une seconde alarme, `MemoryAlert`, est programmée pour faire une mesure toutes les 10 minutes de la quantité de mémoire disponible. Lorsque celle-ci devient inférieure à un certain seuil, un mail est envoyé à l'administrateur. Lorsque 10 mails d'alarme ont été envoyés, l'alarme est coupée.

Une archive zip contenant le code de cet exemple est fournie à la fin de l'article (les sources sont dans le jar).

3.1. Mesure de la consommation mémoire en Java

La mémoire disponible est accessible en appelant la méthode freeMemory() de la classe `java.lang.Runtime`. Néanmoins, cette méthode ne fournit pas forcement un état réel de la mémoire disponible car des objets obsolètes mais qui n'ont pas encore été supprimés peuvent être comptabilisés. Pour obtenir une mesure plus réaliste, il est donc nécessaire de déclencher le garbage collector (GC) et de faire la mesure ensuite. JPlatform fournit la méthode Util.forceFullGarbageCollection() pour forcer le déclenchement du GC.

3.2. Suivi de la consommation mémoire

La classe `MemoryInfo` implémente `AlarmListener` et déclenche l'envoi de mail à chaque fois que sa méthode `handleAlarm()` est appelée.

```
package com.jalios.jcmsplugin.memoryalert;

import java.util.Locale;
import org.apache.log4j.Logger;
import com.jalios.jcms.Channel;
import com.jalios.jdring.AlarmEntry;
import com.jalios.jdring.AlarmListener;
import com.jalios.util.MailUtil;
import com.jalios.util.Util;

public class MemoryInfo implements AlarmListener {

  private static final Logger logger = Logger.getLogger(MemoryInfo.class);

  protected Channel channel;
  protected Locale locale;

  public MemoryInfo() {
    channel = Channel.getChannel();
    locale = new Locale(channel.getDefaultAdmin().getLanguage(), "");
  }

  public void handleAlarm(AlarmEntry entry) {
    try {
      String email = channel.getDefaultAdmin().getEmail();
      logger.info("Send the mail to " + email);
      MailUtil.sendMail(email, email, getMailTitle(), getMailContent());
    }
    catch(Exception ex) {
      logger.warn("An exception occured while handling alarm", ex);
    }
  }

  /**
   * @return the amount of free memoy
   */
  public long getFreeMem() {
    Util.forceFullGarbageCollection(1000);
    return Runtime.getRuntime().freeMemory();
  }

  /**
   * @return the title of the mail
   */
  protected String getMailTitle() {
    return getChannelName() + " Memory Info";
  }

  /**
   * @return the content of the mail
   */
  protected String getMailContent() {
    return
```

```
      "FreeMem: " + Util.formatFileSize(getFreeMem(), locale) + " / " +
      Util.formatFileSize(Runtime.getRuntime().totalMemory(), locale);
    }

    /**
     * @return the name of the channel
     */
    protected String getChannelName() {

      String name = channel.getName();

      if (channel.isJSyncEnabled()) {
        name += " - " + channel.getUrid();
      }

      return "[" + name + "]";
    }
  }
```

3.3. Déclenchement des alertes mémoire

La classe `MemoryAlert` dérive de la précédente. Elle ne déclenche l'alarme que si la mémoire disponible est inférieure à 10 Mo. A chaque alerte, le compteur `alertCount` est incrémenté. Au dixième envoi, l'alarme est déprogrammée. Les méthodes `getMailTitle()` et `getMailContent()` sont surchargées pour spécialiser le mail envoyé.

```
package com.jalios.jcmsplugin.memoryalert;

import org.apache.log4j.Logger;
import com.jalios.jdring.AlarmEntry;
import com.jalios.util.Util;

public class MemoryAlert extends MemoryInfo {

  private static long MIN_MEMORY = 10*1024*1024;
  private static int MAX_ALERT   = 10;

  private int alertCount = 0;

  private static final Logger logger = Logger.getLogger(MemoryAlert.class);

  public void handleAlarm(AlarmEntry entry) {

    // Check if memory is low
    long freeMem = getFreeMem();
    if (freeMem > MIN_MEMORY) {
      return;
    }

    // Send the alarm
    alertCount++;
    logger.info("[MemoryAlert] send an alert !");
    super.handleAlarm(entry);

    // Remove the alarm if it has been sent too many times
    if (alertCount >= MAX_ALERT) {
      logger.info("[MemoryAlert] " + alertCount + " alerts have been sent. Remove alarm for Mer
      entry.isRepetitive = false;
    }
  }

  public String getMailTitle() {
    return getChannelName() + " Memory Alert";
  }

  public String getMailContent() {
    return "FreeMem: " + Util.formatFileSize(getFreeMem(), locale) + "\n" +
           "Alert count: " + alertCount + "/" + MAX_ALERT;
  }

}
```

3.4. Enregistrement des alarmes

La classe `MemoryAlertChannelListener` se charge d'enregistrer les alarmes au démarrage de JPlatform.

```
package com.jalios.jcmsplugin.memoryalert;

import com.jalios.jcms.Channel;
import com.jalios.jcms.ChannelListener;
import com.jalios.jcms.plugin.Plugin;
import com.jalios.jcms.plugin.PluginComponent;
import com.jalios.jdring.AlarmEntry;
import com.jalios.jdring.AlarmListener;
import com.jalios.jdring.AlarmManager;

public class MemoryAlertChannelListener extends ChannelListener implements PluginComponent {

  private static final String SCHEDULE_1      = "30 09 * * * *"; // Every day at 09:30
  private static final String SCHEDULE_2      = "30 14 * * * *"; // Every day at 14:30
  private static final int    ALERT_FREQUENCY = 10; // minutes

  public boolean init(Plugin plugin) {
    return true;
  }

  public void initAfterStoreLoad() throws Exception {
    try {
      AlarmManager alarmMgr = Channel.getChannel().getCommonAlarmManager();
      AlarmListener listener = new MemoryInfo();
      alarmMgr.addAlarm(new AlarmEntry(SCHEDULE_1, listener));
      alarmMgr.addAlarm(new AlarmEntry(SCHEDULE_2, listener));
      alarmMgr.addAlarm(ALERT_FREQUENCY, true, new MemoryAlert());
    }
    catch(com.jalios.jdring.PastDateException ex) {
      ex.printStackTrace();
    }
  }

  public void handleFinalize() {
    // EMPT
  }

  public void initBeforeStoreLoad() throws Exception {
    // EMPTY
  }
}
```

3.5. Déclaration du plugin

Ces trois classes sont déclarées dans le fichier `plugin.xml` .

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plugin PUBLIC "-//JALIOS//DTD JCMS-PLUGIN 1.0//EN" "http://support.jalios.com/dtd/jcr

<plugin name="MemoryAlert" version="0.1" author="Olivier Dedieu" license="Jalios SA" initializ

  <label xml:lang="en">Memory Alert</label>
  <label xml:lang="fr">Memory Alert</label>

  <java-classes>
    <java class="com.jalios.jcmsplugin.memoryalert.MemoryAlert" />
    <java class="com.jalios.jcmsplugin.memoryalert.MemoryInfo" />
  </java-classes>

  <plugincomponents>
    <channellistener class="com.jalios.jcmsplugin.memoryalert.MemoryAlertChannelListener" />
  </plugincomponents>

</plugin>
```

4. Quelques remarques

Dans un environnement répliqué avec JSync, il peut être nécessaire de centraliser le déclenchement des alarmes sur le leader. Pour cela, on testera la méthode `channel.isMainLeader()` avant d'enclencher les alarmes.

Il est possible de combiner `AlarmListener` et `StoreListener`. On peut ainsi encapsuler dans la même classe l'enclenchement d'une alarme lorsqu'un objet créé ou modifié correspond aux critères requis pour l'alarme.

5. Limites et Bonnes pratiques

Il y a quelques limites qu'il faut impérativement comprendre pour bien utiliser les alarmes.

5.1. Un AlarmManager exécute une seule alarme à la fois

Si une alarme *a1* (rattaché à un `AlarmManager` *AM1*), est en cours d'exécution, toutes les autres alarmes du même `AlarmManager` *AM1* devront patienter que *a1* soit fini avant de pouvoir être invoqué.

Conséquence : Une alarme peut décaler dans le temps l'exécution des autres alarmes d'un même `AlarmManager`

Bonne pratique #1 : Il ne faut pas que trop d'alarmes différentes soit rattachées à un seul et même `AlarmManager` sous peine de perturber leur fonctionnement.

Notamment, dans JPlatform, il y a le *CommonAlarmManager*, c'est un alarmManager fourre-tout, que l'on doit utiliser uniquement pour toute les alarmes "mineures" qui ne prennent pas de temps. Sous risque de décaler les autres alarmes de ce manager.

5.2. L'ajout d'une alarme est bloquant si une alarme est en cours d'exécution

Si une alarme *a1* quelconque est en cours d'exécution par l'AlarmManager *AM1*, alors tout invocation de `AM1.addAlarm(a2)` dans un autre thread sera bloquée tant que l'exécution d'alarme *a1* n'est pas terminée.

Conséquence : Lors de l'ajout d'une nouvelle alarme dans un `AlarmManager` existant, il est possible que vous soyez bloqué si cet AlarmManager est déjà occupé. Et inversement si votre alarme est trop longue, vous pourriez bloquer l'ajout d'une alarme
Cette limite est à l'origine du bug JCMS-2891, dans lequel le démarrage de JPlatform se retrouve bloqué pendant un certain temps car une alarme en cours d'exécution dans le *CommonAlarmManager* empêche l'ajout d'une autre alarme dans la suite du démarrage.

Bonne pratique #2 :

- Pour ne pas être bloqué lors de l'ajout : choissisez bien votre `AlarmManager` ,
- Pour ne pas bloquer les autres : Lors du démarrage de JPlatform, ne planifiez pas immédiatement dans le *CommonAlarmManager* des alarmes dont l'exécution pourrait prendre du temps. Sans cela votre alarme pourrait s'exécuter alors que le démarrage est toujours en cours dans le thread principal, et si une autre alarme doit être planifié elle devra vous attendre!

5.3. Un AlarmManager == Un thread

Les `AlarmManager` construits par la methode `channel.getAlarmManager(String key)` créée immédiatement un thread *daemons* qui ne perdurent pas lorsque un site JPlatform est arrêté ou redémarré.

Conséquence : Un `AlarmManager` dédié c'est bien pour gérer une alarme de façon totalement isolée, sans être impacté pas d'autre alarmes et sans impacter par les autres, mais c'est aussi coûteux car cela occupe un thread Java pour cela.

Bonne pratique #3 : Créez de nouveaux `AlarmManager` avec parcimonie.